```haskell
module Region where
type Point = (Double, Double)
type Region   = Point -> Bool
type Radius   = Double
rectangle :: Double -> Double -> Region
rectangle w h = \(x, y) -> and [x >= -w', x <= w', y >= -h', y <= h']
where
 w' = w / 2
 h' = h / 2
inside :: Point -> Region -> Bool
p `inside` r = r p
--------------------------------------------------------------------------------
-- For Pass
dist :: Point -> Point -> Double
dist (px, py) (qx, qy) = sqrt ((qx - px)^2 + (qy - py)^2)
circle :: Radius -> Region
circle r = \p -> dist (0, 0) p <= r
outside :: Region -> Region
outside r = \p -> not (p `inside` r)
combine :: Region -> Region -> Region
r `combine` s = \p -> r p && s p  -- The task was not clear on how to combine, it
is also OK to define it as an or
data Proximity = Close | Near | Far deriving (Eq, Ord, Show)
proximity :: Point -> Point -> Proximity
proximity from to
 | d > 2     = Far
 | d > 1     = Near
 | otherwise = Close
where
 d = dist from to
--------------------------------------------------------------------------------
-- For excellent
move :: Point -> Region -> Region
move (dx, dy) r = r . \(x, y) -> (x - dx, y - dy)
annulus :: Radius -> Radius -> Region
annulus r s | r <= s    = outside (circle r) `combine` circle s
           | otherwise = error "annulus: second radius should be larger"
redCircle, greenAnnulus, blueThing :: Region
redCircle    = move (2, 2)   $ circle 3
greenAnnulus = move (-5, 5)  $ annulus 2 1
blueThing    = move (-2, -4) $ rectangle 10 4 `combine` move (4, 0) (circle 2)
```

```haskell
module Keith where
--------------------------------------------------------------------------------
-- For pass:
digits :: Int -> Int -> [Int]
digits base n = case n `divMod` base of
 (0, r) -> [r]
 (q, r) -> r : digits base q
isKeith :: Int -> Bool
isKeith = isKeithBase 10
--------------------------------------------------------------------------------
-- For excellent:
isKeithBase :: Int -> Int -> Bool
isKeithBase base n = go (digits base n)
where
 go xs | n < base  = False   -- that is length xs < 2
       | m < n     = go (m : init xs)
       | otherwise = m == n
  where
   m = sum xs
keithNumbers :: [Int]
keithNumbers = filter isKeith [1..]
```

```
module Battle where
import Data.List (partition)
import Test.QuickCheck
import Region
--------------------------------------------------------------------------------
-- For pass:
data Piece = Piece { description :: String, region :: Region }
instance Show Piece where  -- This was quite confusing for many, we take it away
from the exam
 show = description
allPieces :: [Piece]
allPieces = [heli, tank, ship]
where
 tank = Piece "Tank" (rectangle 2 3)
 ship = Piece "Ship" (rectangle 3 6)
 heli = Piece "Heli" (circle 2)
movePiece :: Point -> Piece -> Piece
movePiece pos (Piece desc r) = Piece desc (move pos r)
fire :: IO Point
fire = do
 putStr "Take a shot! Please give the x-coordinate:\n> "
 x <- readLn
 putStr "And now the y-coordinate:\n> "
 y <- readLn
 return (x, y)
--------------------------------------------------------------------------------
-- For excellent:
genPiece :: Gen Piece
genPiece = let g = choose (-10, 10) in do
 x <- g
 y <- g
 p <- elements allPieces
 return $ movePiece (x, y) p
play :: Int -> [Piece] -> IO [Piece]
play 0 ps = return ps
play _ [] = return []
play n ps = do
 shot <- fire
 let (hits, ps') = partition (\p -> shot `inside` region p) ps
 print hits
 play (n - 1) ps'
--------------------------------------------------------------------------------
```

```haskell
module URL where
import Test.QuickCheck
input, output :: String
input = "http://alex.nl/age?input=25"
output = "http%3A%2F%2Falex.nl%2Fage%3Finput%3D25"
--------------------------------------------------------------------------------
-- For pass:
escape :: Char -> String
escape c = case c of
 ':' -> "%3A"
 '/' -> "%2F"
 '?' -> "%3F"
 '=' -> "%3D"
 _   -> [c]
type URL = String
encodeURL :: URL -> URL
encodeURL []     = []
encodeURL (x:xs) = escape x ++ encodeURL xs
--------------------------------------------------------------------------------
-- For excellent:
encodeURL' :: URL -> URL
encodeURL' = concatMap escape
-- Some properties, many more are possible
prop_model :: URL -> Bool
prop_model url = encodeURL url == encodeURL' url
prop_safe :: URL -> Bool
prop_safe = all safe . encodeURL
where
 safe x = x `notElem` ":/?="
prop_idempotent :: URL -> Bool
prop_idempotent url = let url' = encodeURL url in url' == encodeURL url'
```